Error Handling

14

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- Error handling.
- Logical errors.
- Syntactical errors such as capitalization, string split, missing or improper imports of classes, and missing curly braces.
- Semantic errors such as improper use of operators, incompatible types, precision, and scoping.
- Importance of error handling with try, catch, and finally blocks.
- Checked verses runtime exceptions.

14.1 | Introduction

Error handling is an important concept in all programming languages. It must be learned and implemented with accuracy to ensure that problems do not ensue in the long run. Even though most people are intimidated by error handling and believe it to be daunting and complicated process, the fact remains that this process is imperative for efficient and effective programming.

Tossed around often, the term "error handling" essentially refers to an entire world of anticipating the possibility of code errors, creating mechanisms to detect them and ultimately, resolving these errors and anomalies in code using various programming and communicating processes. To help put things into perspective, we will first describe error handling from a programmer's point of view to understand how significant error handling is actually for creating and implementing a well-structured program or piece of code.

Next, we will shed some light on the different terms that are involved in error handling and exception handling situations including try, catch, and finally. Then, we will differentiate between runtime and compilation errors. We will also explain how and why the exceptions and errors in both cases vary so greatly, especially in Java programs.

Towards the end of our discussion, we will also shed some light on the different techniques that can be used for error and exception handling. We will also explain how each type of error or exception should be handled based on the situation. This will help programmers get a better understanding of how problems, inaccuracies, and anomalies in the code or program should be dealt with in a manner that solves the problem effectively and efficiently as possible.

14.2 | Understanding Error Handling

Error handling involves a lot more than just the removal of errors. In fact, error handling refers to the entire system of procedures, techniques, and processes that are required to anticipate and identify where problems lie before we get to deal with an error or an exception in the code itself.

Anticipating and knowing that there is a perpetual possibility of errors or exceptions manifesting themselves at any place in your code is the first step involved in effective error and exception handling. This is because once you begin to write code keeping this fact in mind, you will naturally begin to incorporate best practices in your code, minimizing the possibility of errors and exceptions altogether. When errors are anticipated and the possibility of their manifestation is kept in mind, chances are that you will also double check your code more often than you otherwise would in order to minimize problems in the future.

Since errors and exceptions can even occur despite checking the code multiple times, it is imperative for programming languages such as Java to have an error handling system in place, allowing developers to implement a systematic approach to resolve them.

Unlike most concepts that deal with either the software or hardware side exclusively, error handling is one of the few concepts that erases the distinguishing line between the two. Error and exception handling does not only help identify both software and hardware problems, but also allows programmers and developers to deal with the problem at hand in an effective manner, ensuring that the functionality of the rest of the program is not compromised. Needless to say, without the right error and exception handling procedures and techniques in check, it would be impossible for programmers and developers to change one block of code without affecting the rest of it in the process.

Fortunately, in today's day and age, programmers have two major options when it comes to error and exception handling. Programmers can either develop codes and programs that allow room for dealing with errors, or they can make use of software and tools available online and elsewhere to handle problems that are caused by errors and exceptions. While the distinction of the type of error is often clear, there are certain cases in which an error may seem to fall either in a number of different categories, or it may be difficult to identify a category at all due to the ambiguous nature of the error or exception. In cases like these, it is suggested that you make use of customized software for the identification of the errors and exceptions.

There are four major categories of all errors in Java and a majority of other programming languages. These categories are logical errors, generated errors, compile-time errors, and run-time errors. Needless to say, there are different techniques and processes that are involved in dealing with errors of each of these different categories. While the errors in certain categories may be solved and avoided altogether with the help of some basic proofreading of the code, other types of errors may require the programmer or developer to identify the problem with the help of test data and deal with it using resolution programs.

Figure 14.1 illustrates the exception hierarchy in Java.

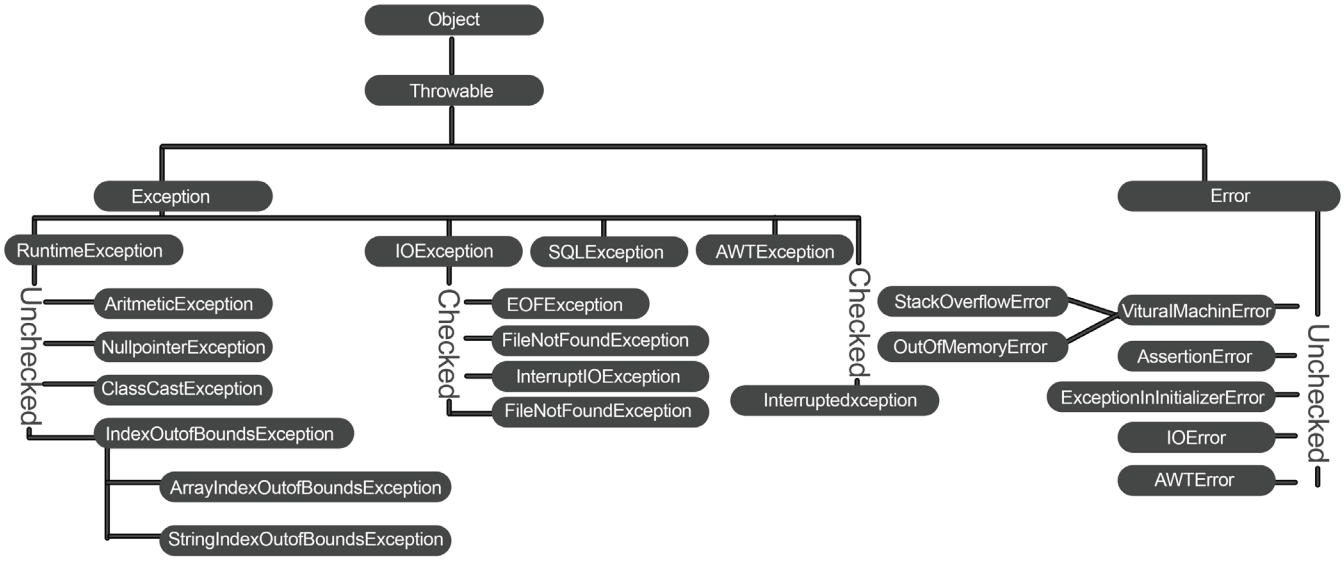

Figure 14.1 Exception hierarchy in Java.

QUICK CHALLENGE Create a chart which shows the difference between Exception and Error. Also provide one example of each.

14.3 | Logical Errors

Logical errors are often considered to be the most difficult types of errors to spot. This is primarily because instead of causing a program to terminate or stop working altogether, logical errors and exceptions produce incorrect results. What this means is that a program with logical errors will run perfectly fine, but you will not be able to see the results which you anticipated, be it due to a typo or incorrect usage in terms of logical operator precedence.

To help you understand better the importance of keeping logical operator precedence in mind, we will illustrate the damage that may arise with the help of the example below. You can see how something as simple as the different usage of parentheses can produce different outputs of 11, 13, 9, and 8.

```
package java11.fundamentals.chapter14;
public class LogicalErrors {
    public static void main(String[] args)
    {
        // Create variables Var1 thru Var4
        int Var1 = 5 + 4 * 3 / 2;
        int Var2 = (5 + 4) * 3 / 2;
        int Var3 = (5 + 4) * (3 / 2);
        int Var4 = (5 + (4 * 3)) / 2;
        // Print the results.
        System.out.println(
            "Var1: " + Var1 +
            "\nnVar2: " + Var2 +
            "\nnVar3: " + Var3 +
            "\nnVar4: " + Var4);
    }
}
```

The above program produces the following result.

```
Var1: 11
nVar2: 13
nVar3: 9
nVar4: 8
```

Since logical errors have more to do with the logic of a program than the actual structure, simple proofreading often suffices for identification and resolution of such errors. While proofreading your code is a practice that should always be implemented, it is essential to double check your code several times especially if it involves a lot of logic building. This is also because logical errors can cause situations and problems that are a lot more severe. In some cases, a condition that is false may even be assumed as true due to incorrect operator usage – something that has the potential to cause errors and problems that will carry on in the rest of the program.

In certain cases, logical errors that have not been spotted at an early stage in the code or program may even have the potential to affect data and values that appear thousands of lines of code after the error was initially made.

One of the most common logical errors that bother programmers is misuse or misplacing of a semicolon. While misplaced semicolons may not give you the results that you had intended, the fact remains that it is completely possible to create a fully functional Java code with misplaced semicolons.

Here is an example of how misplaced semicolons work:

```
package java11.fundamentals.chapter14;
public class ErrorForLoop {
    public static void main(String[] args)
    {
        // Variable Declaration.
        int Counter;
        // Create For Loop.
        for (Counter = 1; Counter <= 10; Counter++)
            ;
        {
            // Print the result.
            System.out.println("Counter is " + Counter);
        }
    }
}
```

The above program produces the following result.

```
Counter is 11
```

In the above example, the only output that will be displayed to the user will be "Counter is 11". This is because the semicolon was placed right after the creation of the *for* loop, instead of placing it at the end of the code block. Had the semicolon been placed at the end of the entire *for* loop block, the output would be a list of individual values for Counter starting from 1 and finishing at 11.

As already mentioned, most logical errors can be removed by proofreading, where you go through each line of code and make sure that all the coding elements are properly set up to avoid such errors. In fact, it is also possible to avoid them in most integrated development environments (IDEs) that provide color recognition of different program elements. Programmers can quickly find out where they may have left a code situation that will result in a logical error.

Techniques that are employed for error handling are often termed as *debugging* or *troubleshooting*. Runtime errors are also quite significant in regard to *debugging* or *troubleshooting*, and they are often handled by setting up countermeasures in the programming environment to avoid such situations altogether. In fact, applications that run on hardware are often designed to have an error handling method, which allows the application to recover from any error and once again restart to provide functionality to the program.

14.4 | Syntactical Errors

Syntactical errors are errors in which the wrong language, or syntax, is used to write a code or program. Not writing the condition in parentheses for an *if* loop, for instance, would classify as a syntax error since that is a requirement for the code to run properly. This even holds true if the condition were to be present on the same line as the *if* statement, but just not in parentheses.

Syntactical errors may not be as difficult to spot as their logical counterparts because the compiler will most likely catch the majority of these errors for you. However, as always, proofreading and knowing the correct syntax and conditions for all loops and codes is essential to ensure that you do not have to face any problems in the long run.

The reason why it is so important to spot and resolve syntactical errors as early as possible is because if a code or program has syntactical errors, it will not be possible for the Java Runtime Environment (JRE) to use the byte code that needs to be created by the compiler. Since syntactical errors have the potential to cause a lot of problems, we are sharing some of the most common syntactical errors and why they are so important to solve.

QUICK CHALLENGE Write a program that can demonstrate syntactical errors.

14.4.1 Capitalization

Java is a case-sensitive language but not a lot of new programmers realize that. This is the major reason why so many new Java programmers start capitalizing keywords instead of writing them in lowercase. Capitalization may or may not make a difference in certain languages; however, in languages such as Java, making a change as apparently insignificant as writing myCount instead of MyCount has the potential to ruin the entire code and fill it with more errors than you ever thought would be possible.

Using the right capitalization is essential for all class names, variable names, and any other piece of code that you will be writing in Java.

14.4.2 Splitting Strings

Dividing your code into a number of lines often does not matter when you are coding in Java. However, there is an exception that applies when you are adding strings in your program or code. Splitting a string so that it comes on more than one line or contains a new line character in it will cause the compiler to throw an exception or object to the code that you have written.

Fortunately, there is a way to create strings that absolutely have to be written in multiple lines. The approach that is required to do this without receiving any errors messages, exceptions, or objections is to add a double quote to the string that appears on

the first line, and then add a plus sign right after this first half of the string ends to show the compiler that whatever follows in the next one or more lines needs to be added or concatenated to the same string. An example of how this can be done is as follows:

```
System.out.print ( "This is the first half of the string " +
"this is the second half of the string that needs to be concatenated. " );
```

14.4.3 Not Importing Classes

One of the most common semantic errors that programmers – both new and seasoned – make when coding in Java is forgetting to include an associated class when they wish to make use of a particular API feature. For instance, if you wish to incorporate the String data type in your code, it is essential to add the class to your application using `Import Java.lang.String;` for the String class to be imported in your application.

14.4.4 Different Methods

When coding in Java, you must remember that static methods and instance methods work differently in this language. Static methods are those that are associated with a specific class, whereas instance methods are associated with the object that is created from a certain class. In case you treat a static method as an instance method in Java, your compiler will present you with a syntactical error since the way in which both of these are dealt with differ greatly.

14.4.5 Curly Braces

When programming in Java, you will often find yourself in situations where you want the same feature to apply to more than one-line code. In cases like these, it is imperative for you to create a block of code enclosed in curly braces to ensure that the compiler understands where the code that the feature needs to be applied to starts and finishes. While the compiler may catch this error for you in most cases, it is still important for you to keep an eye out for lines and blocks of code that need to be treated as a single entity and make sure that they are distinctly identifiable by the compiler.

For instance, if you forget to finish the contents of a class with a curly braces, this will be treated as a syntactical error by the compiler and you will be notified of a missing curly braces. In the example below, the class Cat will not be recognized because the compiler will not know where it ends.

```
package java11.fundamentals.chapter14;
public class Cat {
    int age;
    String breed;
    String color;

    void meowing() { }

    void sleeping() { }

    void hungry() { }
}
```

As seen in the above example, each of the methods – namely meowing, sleeping, and hungry – do not have any contents in them but they still have curly braces to show where they start and end. The class Cat, on the other hand, does not end with a curly braces which is why an error or exception will be generated.

Moreover, in the scenario mentioned above, while the compiler knows that a curly braces is missing, it may not be able to pinpoint the exact location where the curly braces should appear. This is because each of the methods may or may not have been a part of the Cat class. This is why you will only be notified that a curly braces is missing but not where it should appear.

Since using a curly braces in Java is the syntactical rule when a feature or action needs to be applied to multiple lines of code, runtime errors can also occur in case you forget to add a curly braces or you add one in the wrong spot. In case you forget to add a curly braces at the end of an *if* statement, the condition will only apply to the line of code that comes immediately after the *if* statement and can potentially cause problems in the way the application or program was intended to run.

14.5 | Semantic Errors

Figuring out the difference between semantic and syntactical errors is one of the biggest problems for both beginners and seasoned programmers of the Java language. While the majority of people tend to classify both semantic errors and syntactical errors in the same category due to ease and convenience, there are certain significant differences between these types of errors. While syntactical errors have to do with the syntax of the code, semantic errors are related to the usage of the code. This means that it is possible for you to have semantic errors in your code even if the syntax is correct.

While you might have probably already guessed this, the most common type of semantic errors are those in which variables are used without proper initialization. You already know that a variable cannot be used or be expected to add value to your code in case there are problems with its declaration or its initialization. Fortunately, these errors will be caught by the compiler in most cases and you will receive a notification about the variable in question.

While the problem with variables is the most common semantic error, there are plenty of others that you should be aware of as a programmer of the Java language. Subsequent subsections discuss some of the most common semantic errors and explain why each of them occurs. In some cases, we may also tell you how they can be solved or avoided altogether.

What is the difference between syntactical errors and semantic errors?

14.5.1 Improper Use of Operators

Sometimes, in case of operators are used improperly on variables, it may give an impression of syntactical error. However, in reality it is more of a semantic error. For example, the increment operator (++), for instance, cannot use Boolean variables and attempting to do so will be a semantic error.

While new versions of Java are able to detect these problems far more easily, finding out exactly why an error message is generated may be a bit of a challenge especially if you are not sure what operators are allowed to be used with what variables.

Another common operator error mistake the use of comparator (==) operator with objects. Since this is not allowed and comparator operators can only be used with primitive types, an error message will be generated showing that the operation intended is not permissible.

14.5.2 Incompatible Types

Whether it is done by accident or simply due to the lack of knowledge, programmers of the Java language often try to use incompatible types together. Needless to say, doing so is a semantic error that may or may not be caught by the compiler.

For instance, if you mistakenly try to assign a float value to an int variable, the compiler will present an error message. However, if you try to assign an int to a float, the compiler will automatically convert the integer value into a float value. This could potentially cause several problems, especially if that conversion was never intended. Additionally, since the programmer or user will not be notified of this conversion as the compiler will practically expect that this was the desired output, it will become almost impossible for the anyone working on the code to find out that the code contains an error.

14.5.3 Precision

While a float variable can be converted to an int variable by applying casting, doing so incorrectly can result in a loss of precision. Additionally, since everything after the decimal will automatically be lost, the precision of the value that you use will be affected along with the results in all of the other lines or blocks of code where the value in question will be used. It is, therefore, recommended that you only use casting when you absolutely must and know that your output can potentially be affected.

14.5.4 Scoping

Scoping is an issue that tends to bother even the most experienced programmers. This is because there are quite a number of rules that define what is and is not allowed within a certain scope. For instance, if you try to declare a private static int variable inside a method, you will receive an error. Instead, you should declare the variable globally so that it can be used properly. Following programs are examples of incorrect and correct ways to declare a private static int variable.

If you have a class called *VariablePrivate* and you declare the private static int variable globally in the class itself, it will work as follows:

```
package java11.fundamentals.chapter14;
public class VariablePrivate {
    // This is the correct way to declare the private static int variable called
intPrivate.
    private static int intPrivate = 5;
    public static void main(String[] args)
    {
        // The contents of the method will go here
        System.out.println("intPrivate value : " + intPrivate);
    }
}
```

The above program produces the following result.

```
intPrivate value : 5
```

On the other hand, if you try to declare the private static int variable called *intPrivate* within the method itself, you will get an error message. Following is the incorrect way of declaring the private static int variable:

```
package java11.fundamentals.chapter14;
public class VariablePrivateIncorrect {
    public static void main(String[] args)
    {
        private static int intPrivate = 5;
    }
}
```

The above program produces the following result.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Illegal modifier for parameter intPrivate; only final is permitted

        at java9.fundamentals.chapter6.VariablePrivateIncorrect.main(VariablePrivateIncorrect.java:10)
```

14.6 | Importance of Error Handling

The importance of error handling can never be emphasized enough. Error handling does not only ensure smooth operation but also guarantees that the code will not malfunction and will provide the desired results.

Since even the simplest of applications are easily a few thousand lines of code long and comprise code written by a number of programmers, it is extremely important to take care of all errors and exceptions as you move forward. This will ensure that errors, exceptions, and other problems of the sort do not carry forward until the end of the program.

14.6.1 Try, Catch, and Finally

Earlier, we talked all about errors and their different types along with ways in which they can be avoided or resolved. By now, you probably already understand the importance of error handling and realize that it is imperative to resolve errors as soon as they are spotted to ensure that they do not continue to cause problems in the rest of the code. While error handling is something that you are probably already aware of, there is another thing that you should probably worry about – exceptions.

So, what exactly are exceptions and how do they work? As the name suggests, exceptions derange the regular flow of the program or code and make it act in a way that it should not. While there is nothing wrong with the syntax of the code in which an exception is being caused, exceptions still cause problems and make your code or application act in a way that is different than what was expected.

In Java, the concept of exceptions is a lot more profound. Error events in Java are wrapped by exceptions that occur within a method. Exceptions in Java do not only contain information about the error that occurred and the type of the error that has manifested itself in the code, but also details the state of the program when the error occurred. Additionally, some other custom details that can help you assess the nature of the error and the impact that it caused on the code may also be included in the exception.

Fortunately for Java programmers, exceptions can point out a multitude of different error conditions that may occur within the code. When talking about Java virtual machine (JVM) errors, exceptions cannot only help indicate OutOfMemory errors and StackOverflow errors, but they can also help the programmer understand Linkage errors and why they occurred within the code along with details about the error itself. Exceptions can also help programmers understand System errors, including FileNotFound exceptions, IOExceptions, and SocketTimeOutExceptions.

The reason why many programmers of Java language are interested in using exceptions is primarily because these allow them to treat the regular flow of a code as a separate entity, unlike error handling. As a result, you do not only get clearer algorithms that are far easier to handle and understand than regular code, but the clutter within the code also decreases significantly, allowing you to be more creative and innovative with your programs and applications.

Can you delay the exception handling further down the method calls?

In programs or pieces of code where an exception is possible, it is recommended that you use a statement that will be able to catch the exception. By doing so, you can prevent the entire program from crashing should the exception occur. One of the most common statements used for this purpose is the "try" statement. By incorporating a try statement in your code, you can be sure that the potential exception will be caught and treated as a block of code that is separate from the remaining program, code, or application that it is a part of. The general form of the try statement is as follows:

```
try
{
// statements that can potentially cause an exception will go here
}
catch (identifier of type of exception)
{
//statements that should be executed if the exception is thrown will go here
}
```

As seen above, the try statement helps you treat a block of code with the potential of an exception as a separate entity to ensure that the rest of the program is not affected. Additionally, it is seen that the try statement also accommodates an alternative statement or block of code that should be executed should the exception be thrown, as seen in the second half of the example above.

Can you capture errors using try-catch?

In try-catch statements, it is possible for you to code multiple try blocks. This particularly comes in handy where the statements in the try block throw exceptions of different types. Additionally, from Java 7 onwards, it is even possible to catch multiple types of exceptions within the same catch block by separating the type using vertical bars. An example of how this is done is as follows:

```
try
{
    // statements that have the potential to throw
    // ClassNotFoundException
    //ArrayIndexOutOfBoundException
}
catch (ArrayIndexOutOfBoundException | ClassNotFoundException e)
{
    System.out.println(e.getMessage());
}
```

It is also important to note that the contents of the try block are not visible to the catch block, which is why it is impossible for any variables declared in the try block to be used in the catch block or blocks. In case there is a variable that you need to use in both blocks of code, it should be declared before the try block.

The following is another example of how the try-catch statement can be used in action to prevent the user from trying to divide the value of a variable by 0.

```
package java11.fundamentals.chapter14;
public class DivideByZero {
    public static void main(String[] args)
    {
        int var1 = 5;
        int var2 = 0; // 0 is assigned to var2 to cause an exception by dividing var1 by 0
        try
        {
            int var3 = var1 / var2; // This is the statement that will cause the exception
to be thrown
        }
        catch (ArithmeticException e)
        {
            System.out.println("It is not possible to divide by zero");
        }
    }
}
```

The above program produces the following result.

```
It is not possible to divide by zero
```

Another block of code that is used for exceptions in Java is the finally block. One unique and interesting feature about the finally block is that it is always executed regardless of whether or not any exceptions have been thrown in the code. With that said, using the finally statement is among best practices and is expected to be used particularly in scenarios when you are closing a connection or file. Here is an example of a program where the finally block will be executed:

```
package java11.fundamentals.chapter14;
public class FinallyBlockExample {
    public static void main(String args[])
    {
        try
        {
            int var = 30 / 6;
            System.out.println(var);
        }
        catch (NullPointerException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("These are the contents of the finally block");
        }
        System.out.println("The finally block has been executed");
    }
}
```

The above program produces the following result.

```
5
These are the contents of the finally block
The finally block has been executed
```

Since no exception will be thrown, the value of the variable "var" will be displayed to the user, after which the finally block will be executed and the words "These are the contents of the finally block" will be displayed. Next, the words "The finally block has been executed" will be displayed to the user.

14.7 | Checked verses Runtime Exceptions

As the name suggests, checked exceptions are those that are identified at the time when the code is being compiled. On the other hand, runtime exceptions are identified when the code is being run.

Can program execution continue even after an exception is thrown?

14.7.1 Checked Exceptions

In case a checked exception is being thrown by a method, it is necessary that this exception should either be handled by the method itself, or the *throws* keyword should be used to specify it. The following example shows how a checked exception is thrown in the `main()` function.

```
package java11.fundamentals.chapter14;
import java.io.*;
public class CheckedExceptionsExample {
    public static void main(String[] args)
    {
        FileReader file = new FileReader("D:\\newfolder\\example.txt");
        BufferedReader fileInput = new BufferedReader(file);
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());
            // This block of code will output the first 3 lines of the file
            // "D:\newfolder\example.txt"
            fileInput.close();
    }
}
```

The above program produces the following result.

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        Unhandled exception type FileNotFoundException
        Unhandled exception type IOException
        Unhandled exception type IOException

        at java9.fundamentals.chapter6.CheckedExceptionsExample.main(CheckedExceptionsExample.java:9)
```

In the example above, the `main()` function uses FileReader to read the file located at D:\\newfolder\\example.txt. (If you are executing this code on your end, make sure you change the file location as per the location of file on your computer). However, a checked *FileNotFoundException* is thrown by `FileReader()`, whereas checked *IOException* is thrown by the `close()` and `readLine()` methods. As a result, a message will be displayed showing where the exception was thrown, explicitly stating that the source code cannot be compiled.

14.7.2 Runtime Exceptions

Runtime exceptions in Java are a subcategory of unchecked exceptions or exceptions that are not checked during the compilation of the code. Since unchecked exceptions and runtime exceptions are not detected at the time of compilation, the code compiles perfectly before the runtime exception is detected when the program is run. The following is an example of a runtime exception:

```
package java11.fundamentals.chapter14;
public class RuntimeExceptionExample {
    public static void main(String args[])
    {
        int var1 = 0;
        int var2 = 10;
        int var3 = var2 / var1;
    }
}
```

The above program produces the following result.

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at java9.fundamentals.chapter6.RuntimeExceptionExample.main(RuntimeExceptionExample.java:9)
```

In the example above, the `main()` function will throw an ArtihmeticException which is a type of runtime exception, as it is not possible to divide by 0. However, since there is nothing wrong with the syntax of the code, it will compile perfectly before the exception is detected when the program is run.

QUICK CHALLENGE Write a program which can demonstrate all types of Runtime Exceptions.

Summary

In this chapter, we have discussed error handling and how to use it. We also discussed the concept of error handling and elucidated various types of errors such as logical, syntactical, and semantic. Then we discussed the importance of error handling in which we studied the use of try, catch, and finally blocks. At the end of the chapter, we learnt the difference between checked and runtime exception with examples.

In this chapter, we have learned the following concepts:

1. What is error handling? What is the importance of using it?
2. What are logical errors, syntactical errors such as capitalization, string split, missing or improper imports of classes, and missing curly braces?
3. What are semantic errors such as improper use of operators, incompatible types, precision, and scoping?
4. How do we handle errors using try, catch, and finally blocks?
5. What is the difference between checked and runtime exceptions?

In Chapter 15, we will learn about garbage collection, using it in a program, and how to implement the garbage collector.

Multiple-Choice Questions

1. Exceptions arises during _____ in the code sequence.
   - **(a)** Compilation time
   - **(b)** Run time
   - **(c)** Can occur anytime
   - **(d)** None of the above
2. _____ is not an exception handling keyword.
   - **(a)** finally
   - **(b)** thrown
   - **(c)** catch
   - **(d)** try
3. Exception can be thrown manually by using _____ keyword.
   - **(a)** finally
   - **(b)** throw
   - **(c)** catch
   - **(d)** try
4. Which of the following is the parent of Error?
   - **(a)** Object
   - **(b)** Collections
   - **(c)** Throwable
   - **(d)** Exception
5. What do you understand by unchecked exceptions?
   - **(a)** Checked by Java virtual machine
   - **(b)** Checked by Java compiler
   - **(c)** (a) and (b)
   - **(d)** None of the above

Review Questions

1. What is error handling?
2. How is error handling useful?
3. What is the difference between error and exception?
4. Which exception is thrown when no class is found?
5. How do try, catch, finally blocks work?
6. What are semantics errors?
7. What are logical errors?

Exercises

1. Write a program that can produce exceptions and catch them using try, catch, finally blocks.
2. Create a comparison chart to distinguish between error and exception.
3. Write a program that can produce errors. Observe and document the outcome.

Project Idea

Create a calculator program that performs various types of arithmetic operations. Also create a program that can divide any number by any number. Make sure you add exception handling to capture and notify users in case they use unpermitted operations, such as dividing a number by 0. Make sure your program has all the features of a normal calculator.

Recommended Readings

1. Oracle Tutorials – https://www.w3schools.com/java/java_try_catch.asp
2. W3Schools – https://docs.oracle.com/javase/tutorial/essential/exceptions/
3. Oracle Technetwork – https://www.oracle.com/technetwork/java/effective-exceptions-092345.html